



plain concepts

PowerShell Core

- Operadores, variables, cadenas, arrays...

OPERADORES, VARIABLES, CADENAS, ARRAYS...

Variables

- Usamos el character \$ para crear y usar variables `$myVariable`
- Puede tener letras, números, espacios y guiones bajos `${My variable}`
- No persiste una vez salimos de la Shell
 - New-Variable
 - Set-Variable
 - Get-Variable
 - Clear-Variable
 - Remove-Variable
- Podemos forzar el tipo – `[int]$var`

plain concepts

Nota: El símbolo \$ no es parte del nombre de la variable, es una marca para acceder al contenido de la variable

Type	Description
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed-length string of Unicode characters
[char]	A Unicode 16-bit character
[byte]	An 8-bit unsigned character
[bool]	Boolean True/False value
[decimal]	An 128-bit decimal value
[single]	Single-precision 32-bit floating point number
[double]	Double-precision 64-bit floating point number
[datetime]	Date and Time
[array]	An array of values
[hashtable]	Hashtable object

OPERADORES, VARIABLES, CADENAS, ARRAYS...

Arrays

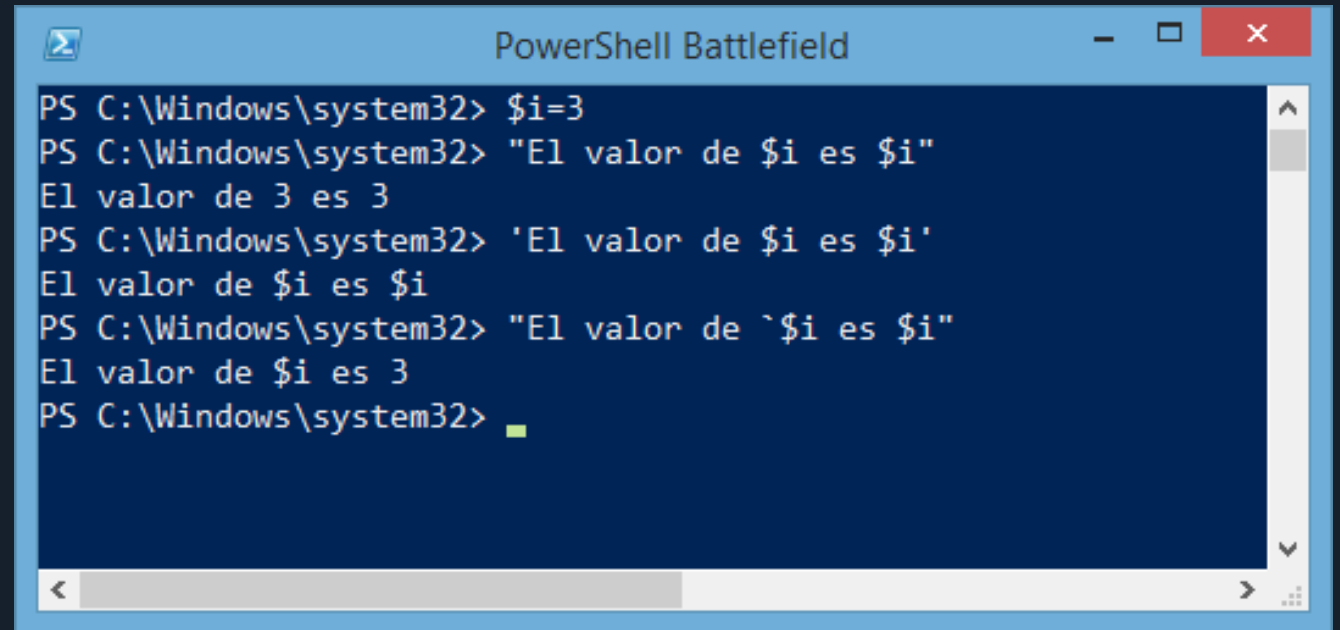
- Array vacío:
 - `$miArray = @()`
- Sintaxis Explícita:
 - `$miArray = @(1,"Hola",3.5,"Mundo")`
- Sintaxis Implícita
 - `$miArray = 1,"Hola",3.5,"Mundo"`
- Operador de rango:
 - `$miArray = (1..7)`
- Añadir Elementos al Array:
 - `$miArray += "NuevoElemento"`

```
PS C:\Windows\system32> $array=1,2,3,4
PS C:\Windows\system32> $array[3]
4
PS C:\Windows\system32>
PS C:\Windows\system32> $array=(1..7)
PS C:\Windows\system32> $array[6]
7
PS C:\Windows\system32>
```

OPERADORES, VARIABLES, CADENAS, ARRAYS...

Cadenas: Uso de las "Comillas"

- Con las dobles comillas se resuelven todas las variables
- Con las comillas simples no hay sustitución
- Get-Help About_Quoting_Rules
- Con el acento grave forzamos la no resolución de variables individuales

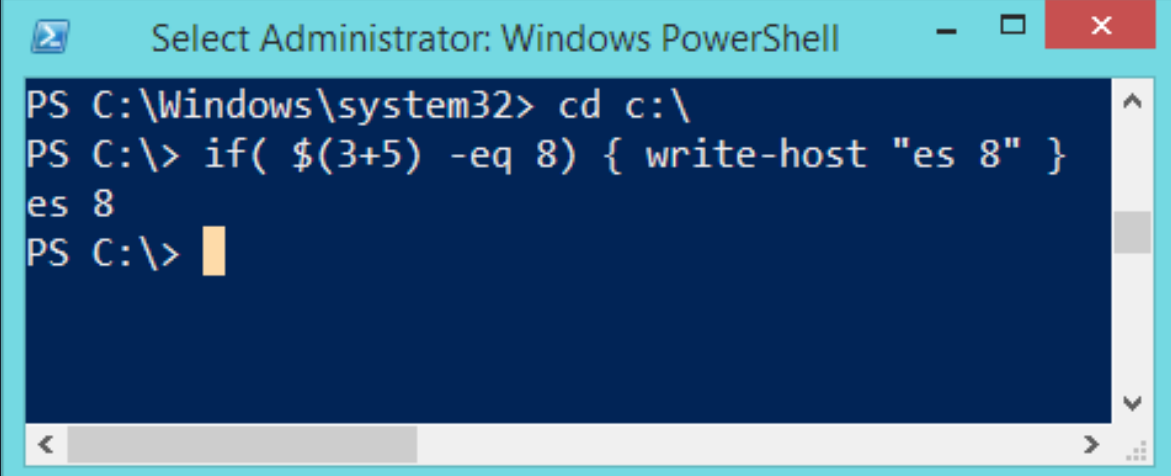


```
PS C:\Windows\system32> $i=3
PS C:\Windows\system32> "El valor de $i es $i"
El valor de 3 es 3
PS C:\Windows\system32> 'El valor de $i es $i'
El valor de $i es $i
PS C:\Windows\system32> "El valor de ` $i es $i"
El valor de $i es 3
PS C:\Windows\system32>
```

OPERADORES, VARIABLES, CADENAS, ARRAYS...

Sub-Expresiones

- Devuelve propiedades específicas de un objeto
- Puede constar de multiples secciones
- Cada sección contribuye al valor devuelto
- Podemos evaluar expresiones en una línea sin necesitar variables intermedias
- Útil para extraer valores de objetos o aplicarles un procesamiento previo a su evaluación

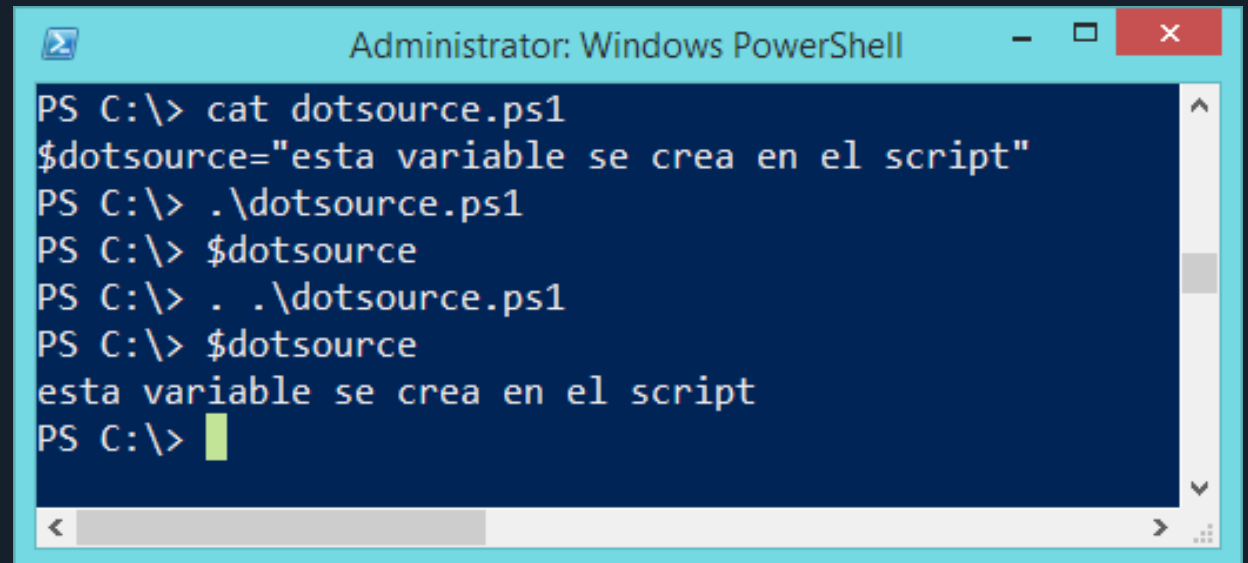


```
PS C:\Windows\system32> cd c:\
PS C:\> if( $(3+5) -eq 8) { write-host "es 8" }
es 8
PS C:\>
```

OPERADORES, VARIABLES, CADENAS, ARRAYS...

DotSourcing

- El operador "." ejecuta un script en el ámbito de ejecución activo.
- Por defecto, cuando se ejecuta un script, éste lo hace en un nuevo ámbito de ejecución.
 - Cuando el script finaliza, las variables creadas desaparecen.
- Al ejecutar scripts en el ámbito actual, las variables creadas permanecen en el ámbito actual



```
Administrator: Windows PowerShell
PS C:\> cat dotsource.ps1
$dotsource="esta variable se crea en el script"
PS C:\> .\dotsource.ps1
PS C:\> $dotsource
PS C:\> . .\dotsource.ps1
PS C:\> $dotsource
esta variable se crea en el script
PS C:\>
```



plain concepts

Proveedores de PowerShell

- Otra forma de interactuar

PROVEEDORES

- Permiten que cualquier tipo de dato sea expuesto a powershell como un Sistema de archivos, como si se hubiese montado una unidad
- Son programas basados en .NET
- Cada proveedor proporciona cmdlets para manejar los datos

Provider	Drive	Data store
-----	-----	-----
Alias	Alias:	Windows PowerShell aliases
Certificate	Cert:	x509 certificates for digital signatures
Environment	Env:	Windows environment variables
FileSystem	*	File system drives, directories, and files
Function	Function:	Windows PowerShell functions
Registry	HKLM:, HKCU:	Windows registry
Variable	Variable:	Windows PowerShell variables
WSMan	WSMan:	WS-Management configuration information

* The FileSystem drives vary on each system.

PROVEEDORES: CERTIFICADOS

```
Administrator: Windows PowerShell

PS HKLM:\SOFTWARE> cd cert:
PS Cert:\> ls

Location      : CurrentUser
StoreNames    : {TrustedPublisher, ClientAuthIssuer, Root, UserDS...}

Location      : LocalMachine
StoreNames    : {TrustedPublisher, ClientAuthIssuer, Remote Desktop, Root...}

PS Cert:\> cd .\LocalMachine\My
PS Cert:\LocalMachine\My> ls

Directory: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint      Subject
-----
E4FE22296833C54A8CAEF98445DFA7448F12722C CN=localhost
CC92344A44139379B7B9831100BE49973483167D CN=AMARQUEZ-HP.plainconcepts.com
```



plain concepts

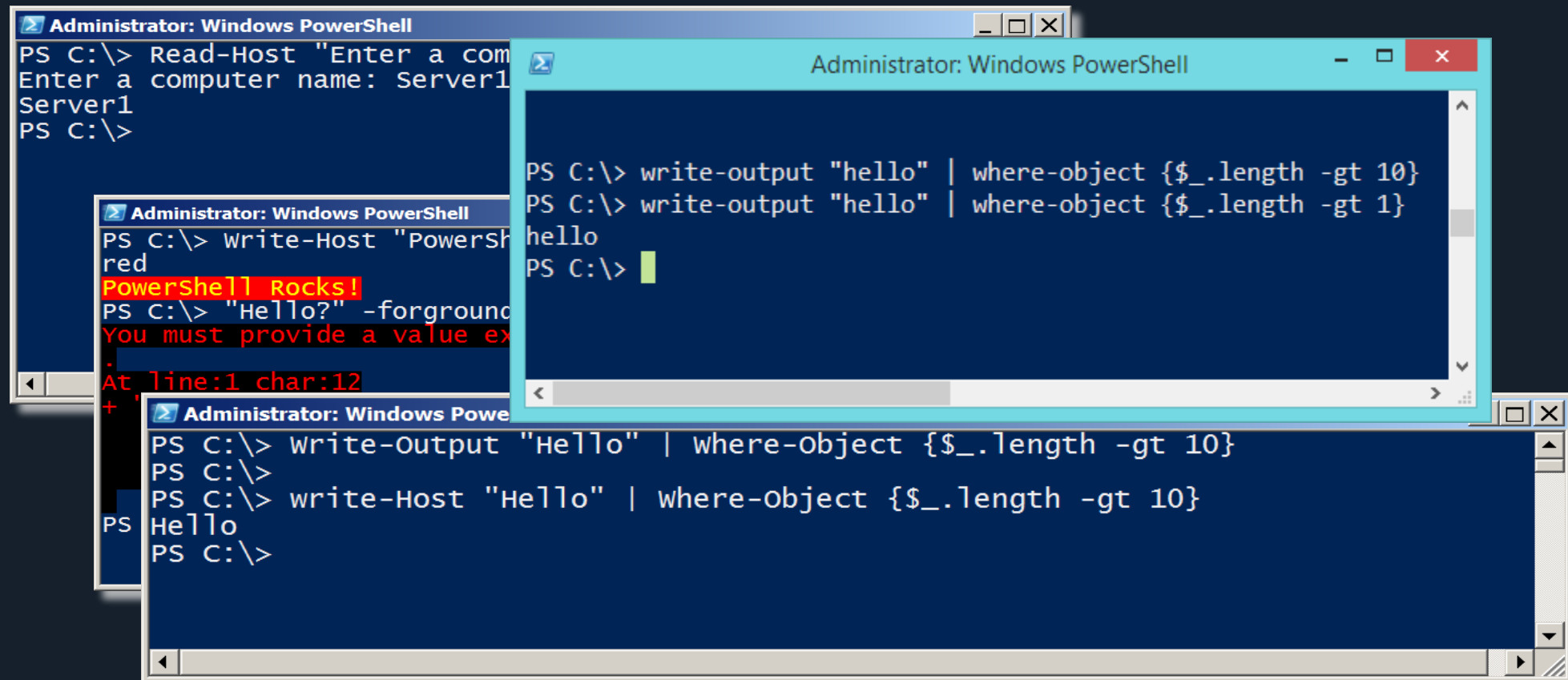
PowerShell Scripting

- Scripting básico

SCRIPTING

Back to basics: Entrada/Salida Interactiva

- Read-Host
- Write-Host
- Write-Output



The image shows three overlapping Windows PowerShell windows. The top-left window demonstrates the `Read-Host` command, which prompts the user to enter a computer name. The top-right window demonstrates the `Write-Output` command, which outputs the word "hello" and then filters it using `where-object` based on its length. The bottom window demonstrates the `Write-Host` command, which outputs the word "Hello" in red text. The bottom window also shows an error message: "You must provide a value ex" and "At line:1 char:12".

```
Administrator: Windows PowerShell
PS C:\> Read-Host "Enter a computer name:"
Enter a computer name: Server1
Server1
PS C:\>
```

```
Administrator: Windows PowerShell
PS C:\> Write-Host "PowerShell Rocks!"
PowerShell Rocks!
PS C:\> "Hello?" -foregroundcolor red
You must provide a value ex
At line:1 char:12
+ ~~~~~
```

```
Administrator: Windows PowerShell
PS C:\> Write-Output "hello" | where-object {$_ .length -gt 10}
PS C:\> Write-Output "hello" | where-object {$_ .length -gt 1}
hello
PS C:\>
```

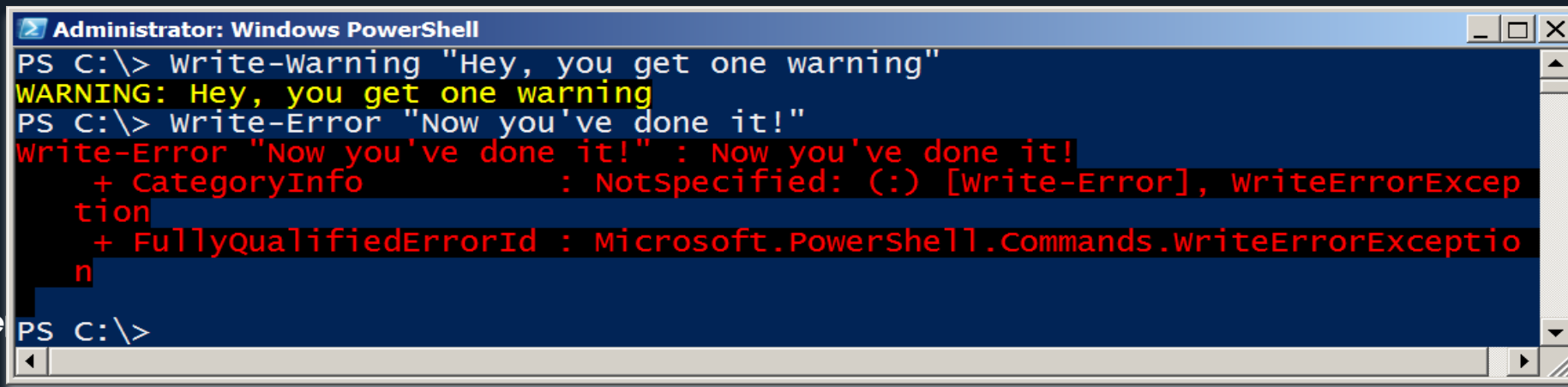
```
Administrator: Windows PowerShell
PS C:\> Write-Output "Hello" | where-object {$_ .length -gt 10}
PS C:\>
PS C:\> Write-Host "Hello" | where-object {$_ .length -gt 10}
PS C:\>
Hello
PS C:\>
```

SCRIPTING

Otras salidas interesantes para scripts y automatización

- Write-Warning
- Write-Verbose
- Write-Debug
- Write-Error

plain conce



```
Administrator: Windows PowerShell
PS C:\> write-warning "Hey, you get one warning"
WARNING: Hey, you get one warning
PS C:\> write-error "Now you've done it!"
write-error "Now you've done it!" : Now you've done it!
+ CategoryInfo          : Notspecified: (:) [write-error], writeErrorExcept
tion
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
PS C:\>
```

SCRIPTING

Objetivos de seguridad de PowerShell

- Por defecto está bloqueada la ejecución de Scripts
- Los archivos .Ps1 se pueden editar con el Bloc de Notas
- Es necesario escribir la ruta para ejecutar un script (./nombreScript.Ps1)

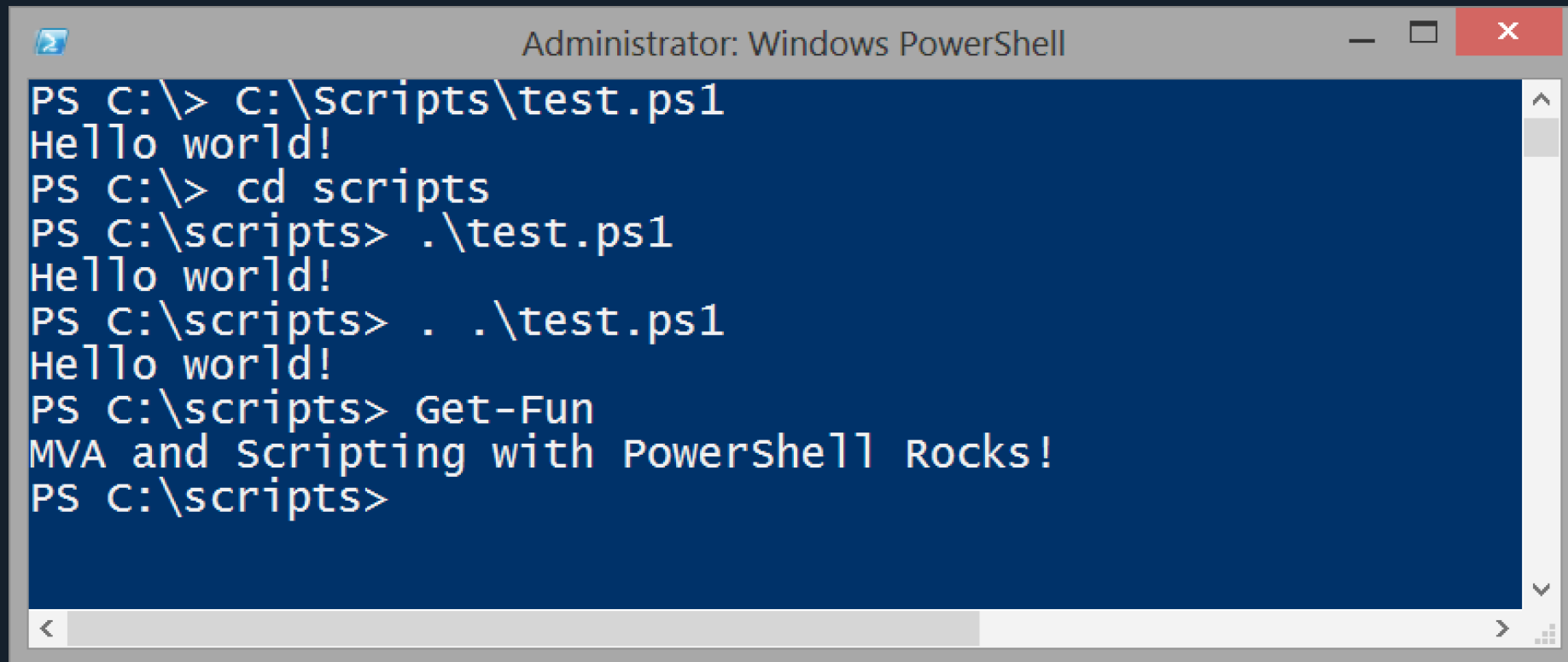
SCRIPTING

Política de ejecución

- Por defecto, PowerShell no permite la ejecución de ningún script.
- Get/Set-ExecutionPolicy
 - Restricted
 - Unrestricted
 - RemoteSigned
 - ...
- Se puede establecer por GPO

SCRIPTING

Ejecutando Scripts

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window has a blue background and a white command line. The command line shows the following sequence of commands and output:

```
PS C:\> C:\scripts\test.ps1
Hello world!
PS C:\> cd scripts
PS C:\scripts> .\test.ps1
Hello world!
PS C:\scripts> . .\test.ps1
Hello world!
PS C:\scripts> Get-Fun
MVA and Scripting with PowerShell Rocks!
PS C:\scripts>
```

The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

plain concepts

DotSourcing!

SCRIPTING

Parametrizando nuestros comandos

```
1 #Choose values to replace with variables
2 |
3 $ComputerName='Client'
4 $Drive='c:'
5
6 Get-WmiObject -class Win32_LogicalDisk -Filter "DeviceID='$Drive'" `
7     -ComputerName $ComputerName
```

- Añade flexibilidad a las acciones ejecutadas.
- Incrementa reusabilidad

SCRIPTING

Object Members y Variables

- Con Get-Member (gm) sacamos el TypeName, Methods y Properties de un Objeto.
- Podemos usar las variables para trabajar con los Objetos.

```
1  #Object Members and variables
2  #Variables are very flexible
3  $Service=Get-Service -Name bits
4  $Service | GM
5  $Service.Status
6  $service.Stop()
7  $Msg="Service Name is $($service.name.ToUpper())"
8  $msg
9  #Working with multiple objects
10 $Services=Get-Service
11 $services[0]
12 $services[0].Status
13 $Services[-1].Name
14 "Service Name is $($services[4].DisplayName)"
15 "Service Name is $($services[4].name.ToUpper())"
```



SCRIPTING

Bifurcaciones

```
1 If ($this -eq $that) {  
2     # commands  
3 } elseif ($those -ne $them) {  
4     # commands  
5 } elseif ($we -gt $they) {  
6     # commands  
7 } else {  
8     # commands  
9 }
```

```
1 # Switch can be easier to maintain than If statement  
2 #and can provide additional features  
3  
4 Switch ($status) {  
5     0 { $status_text = 'ok' }  
6     1 { $status_text = 'error' }  
7     2 { $status_text = 'jammed' }  
8     3 { $status_text = 'overheated' }  
9     4 { $status_text = 'empty' }  
10    default { $status_text = 'unknown' }  
11 }  
12
```



SCRIPTING

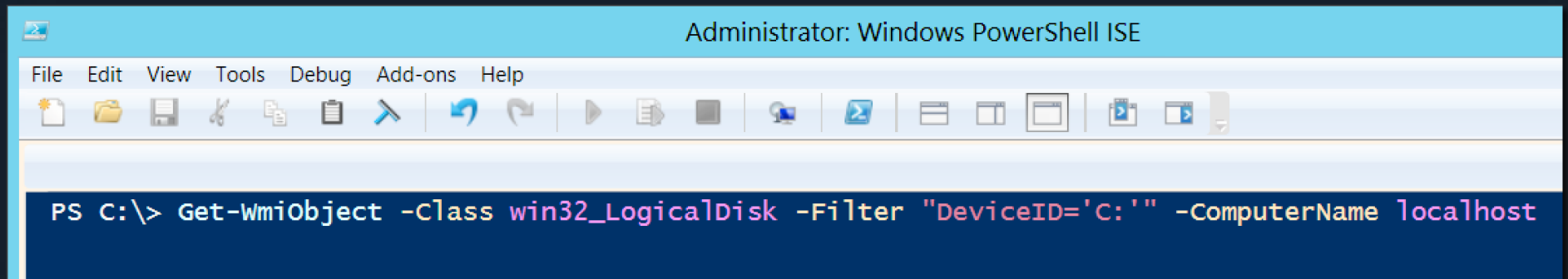
Bucles

```
1  # Do loop
2  $i= 1
3  Do {
4      Write-Output "PowerShell is Great! $i"
5      $i=$i+1 # $i++
6  } while ($i -le 5) #Also Do-Until
7
8  # while Loop
9  $i=5
10 while ($i -ge 1) {
11     Write-Output "Scripting is great! $i"
12     $i--
13 }
14
```

```
1  # Foreach - used often in our scripting for today
2  $services = Get-Service
3  Foreach ($service in $services) {
4      $service.Displayname
5  }
6
7  #For loop
8  For ($i=0;$i -lt 5;$i++) {
9      #do something
10 }
11
12 #Another way
13 1..5 | ForEach-Object -process {
14     Start calc
15 }
16
```

SCRIPTING

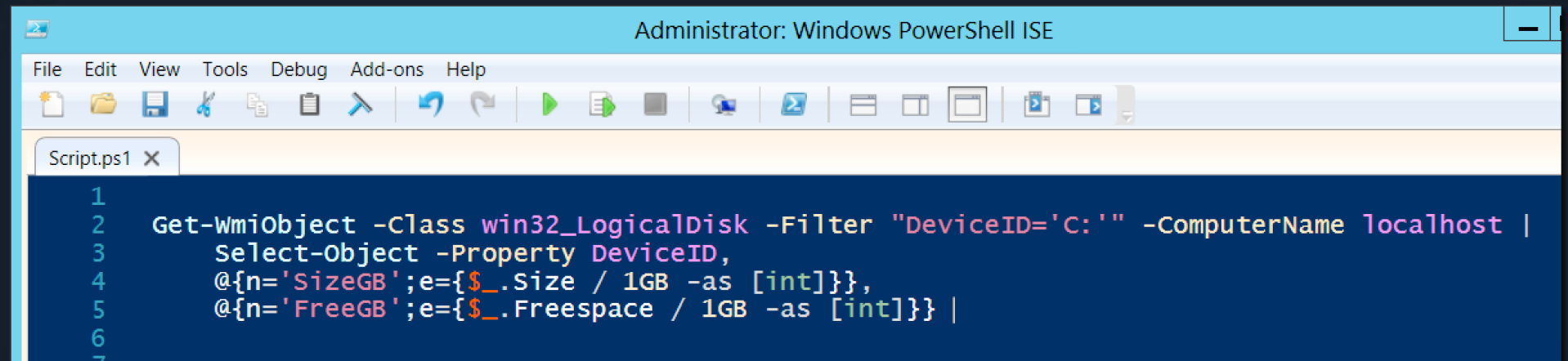
La ISE (Integrated Scripting Environment)



Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

```
PS C:\> Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName localhost
```



Administrator: Windows PowerShell ISE

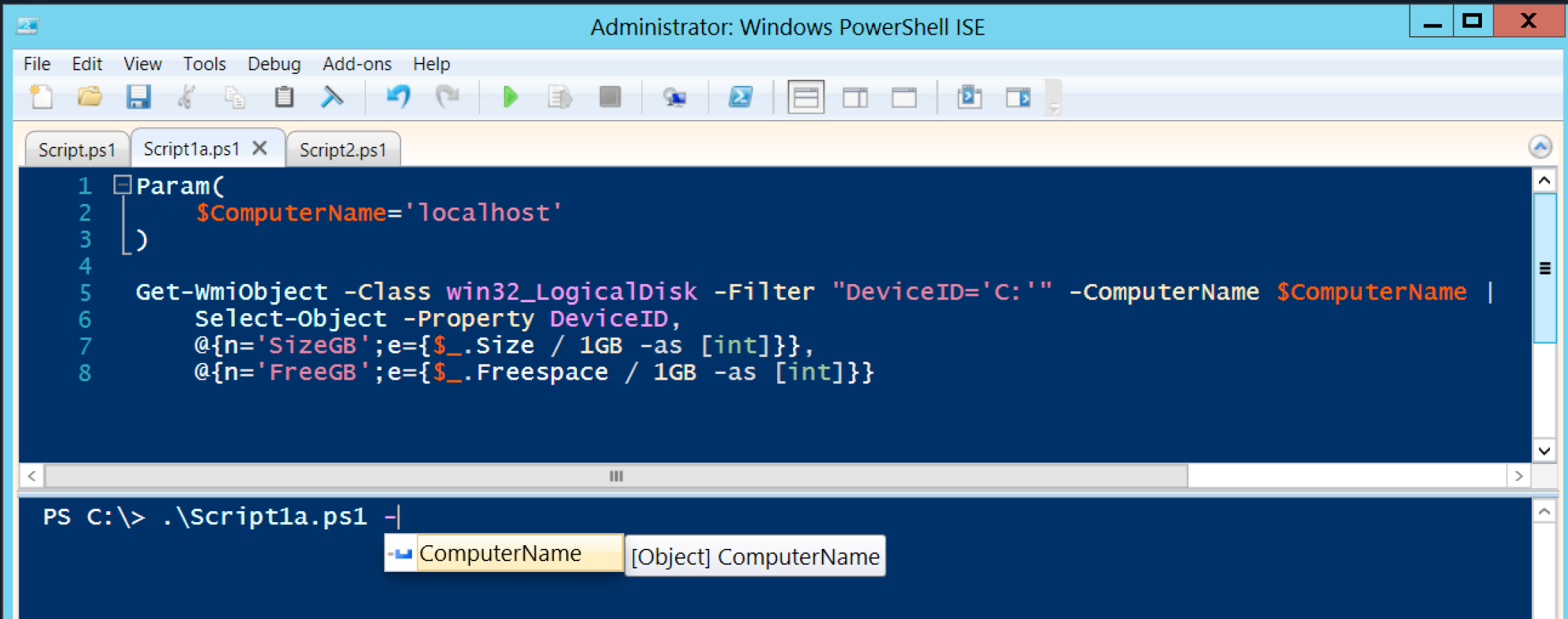
File Edit View Tools Debug Add-ons Help

Script.ps1 X

```
1  
2 Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName localhost |  
3   Select-Object -Property DeviceID,  
4     @{n='SizeGB';e={$_.Size / 1GB -as [int]}} ,  
5     @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}} |  
6  
7
```

SCRIPTING

Añadiendo parámetros a nuestro Script



The screenshot shows the Windows PowerShell ISE interface. The title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations and execution. The script editor shows a file named "Script1a.ps1" with the following code:

```
1 Param(  
2     $ComputerName='localhost'  
3 )  
4  
5 Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName $ComputerName |  
6     Select-Object -Property DeviceID,  
7     @{n='SizeGB';e={$_.Size / 1GB -as [int]}},  
8     @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}}
```

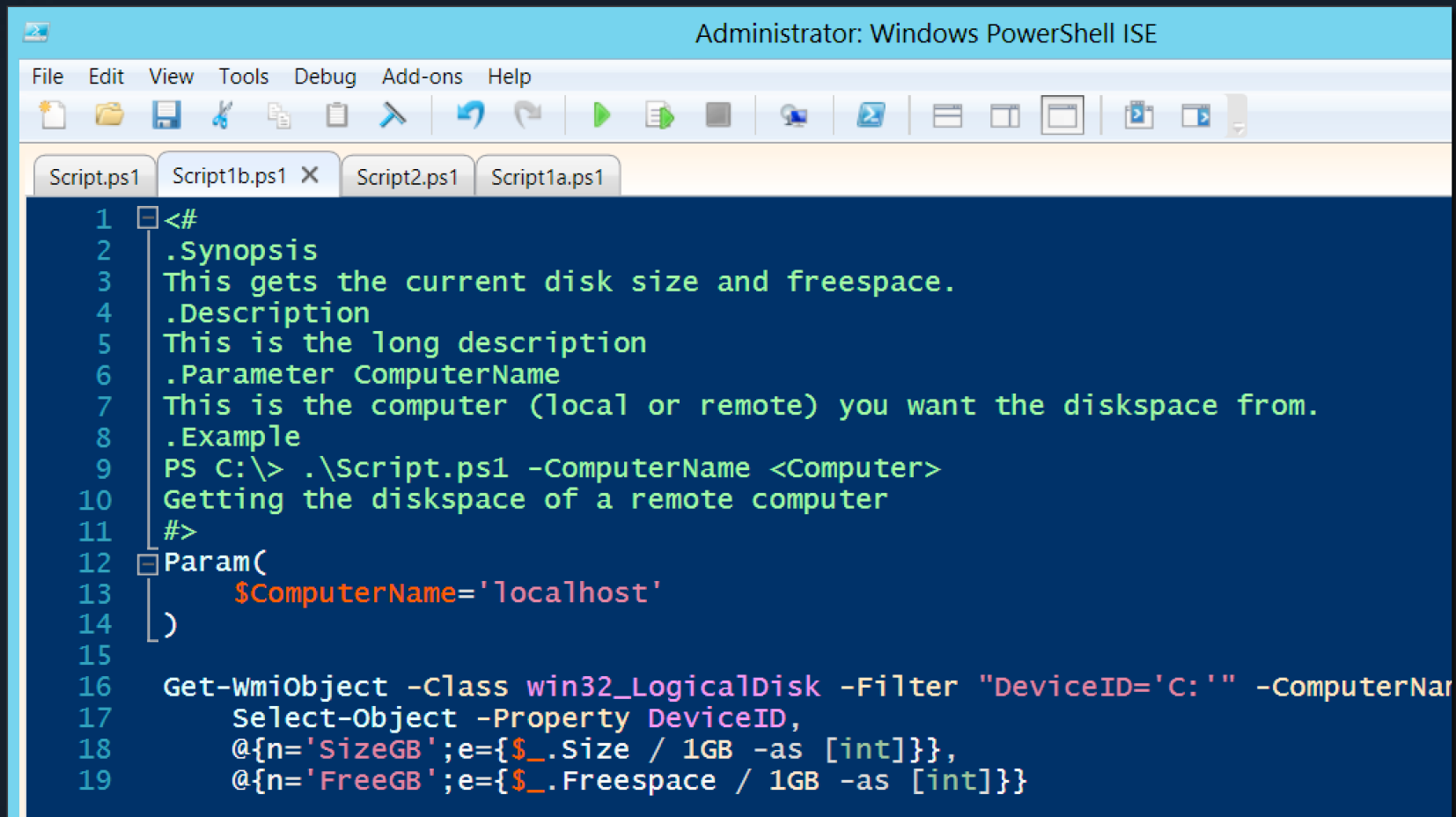
The console window at the bottom shows the command `PS C:\> .\Script1a.ps1 -` and the output:

```
ComputerName [Object] ComputerName
```

SCRIPTING

Documentando el Script

- Comentarios de código
- Se comporta como un bloque único de comentario
- Tiene varias secciones
 - .Synopsis
 - .Description
 - .Parameter
 - ...
- Se integra totalmente con Get-Help



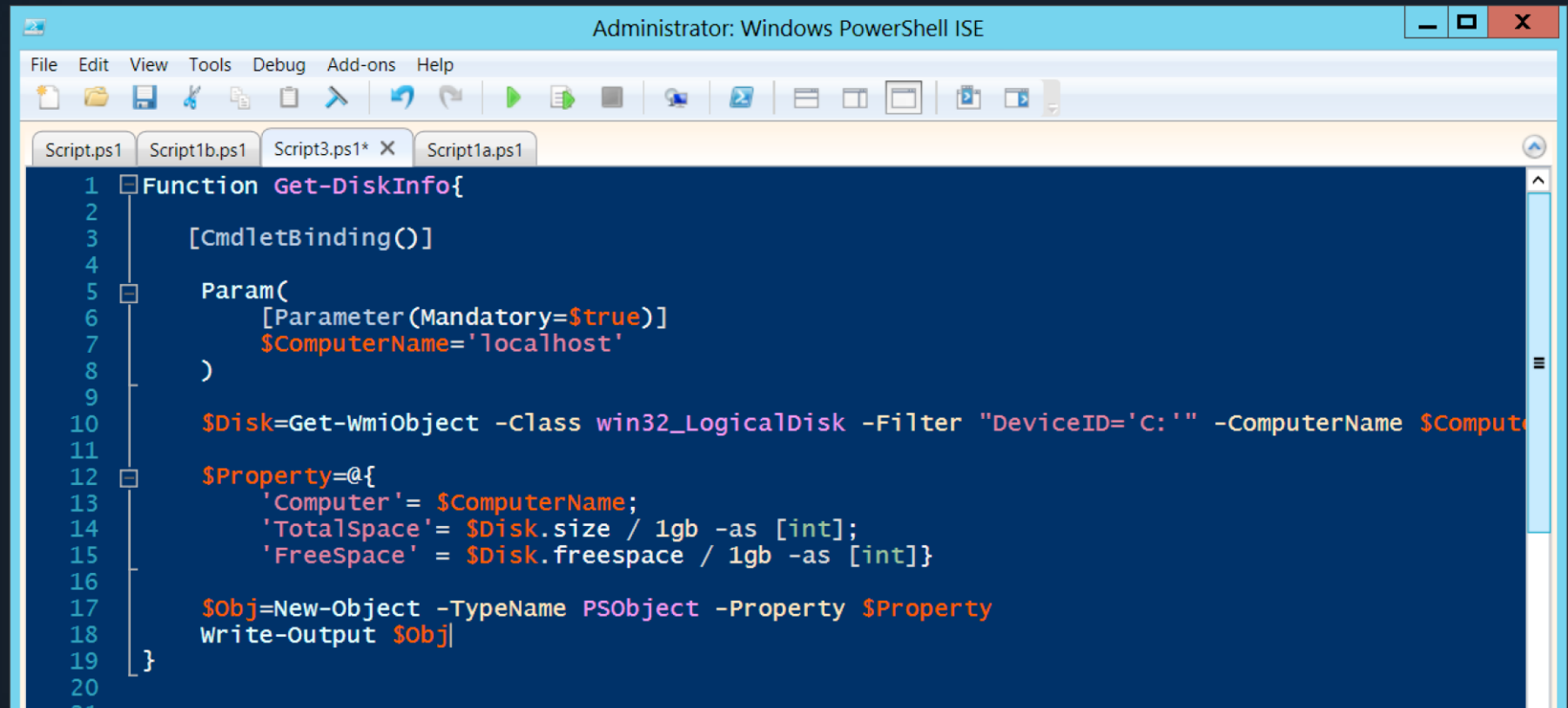
The screenshot shows the Windows PowerShell ISE interface. The title bar reads "Administrator: Windows PowerShell ISE". The menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations and execution. The script editor shows a file named "Script.ps1" with the following content:

```
1 <#  
2 .Synopsis  
3 This gets the current disk size and freespace.  
4 .Description  
5 This is the long description  
6 .Parameter ComputerName  
7 This is the computer (local or remote) you want the disk space from.  
8 .Example  
9 PS C:\> .\Script.ps1 -ComputerName <Computer>  
10 Getting the disk space of a remote computer  
11 #>  
12 Param(  
13     $ComputerName='localhost'  
14 )  
15  
16 Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName  
17     Select-Object -Property DeviceID,  
18     @{n='SizeGB';e={$_.Size / 1GB -as [int]}},  
19     @{n='FreeGB';e={$_.Freespace / 1GB -as [int]}}
```

SCRIPTING

Transforma tu script en una función

- Modulariza la funcionalidad.
- [CmdletBinding()] indica que la función se comporte como un cmdlet



```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script.ps1 Script1b.ps1 Script3.ps1* X Script1a.ps1
1 Function Get-DiskInfo{
2
3     [CmdletBinding()]
4
5     Param(
6         [Parameter(Mandatory=$true)]
7         $ComputerName='localhost'
8     )
9
10    $Disk=Get-WmiObject -Class win32_LogicalDisk -Filter "DeviceID='C:'" -ComputerName $ComputerName
11
12    $Property=@{
13        'Computer' = $ComputerName;
14        'TotalSpace' = $Disk.size / 1gb -as [int];
15        'FreeSpace' = $Disk.freespace / 1gb -as [int]}
16
17    $Obj=New-Object -TypeName PSObject -Property $Property
18    Write-Output $Obj
19 }
20
21
```

SCRIPTING

Documentando tu función

```
function Get-SecureString (){  
    <#  
    .SYNOPSIS  
    Lee una contraseña desde teclado y la devuelve como un SecureString  
  
    .DESCRIPTION  
    Lee una contraseña desde teclado y la devuelve como un SecureString  
  
    .NOTES  
    #>  
        return read-host -Prompt "Password" -AsSecureString | ConvertFrom-SecureString  
    }
```




plain concepts

Gestión de Errores

GESTIÓN DE ERRORES

Control de errores

- Cuando un commando de PowerShell encuentra un error que no termina la ejecución, comprueba el valor de la variable por defecto **\$ErrorActionPreference**:
 - Continue – Es el valor por defecto. Muestra el error y sigue ejecutándose.
 - SilentlyContinue – Sigue ejecutándose pero no muestra el error.
 - Stop – Muestra el error y para la ejecución.
 - Inquire – Muestra un mensaje para que el usuario elija que acción tomar.

GESTIÓN DE ERRORES

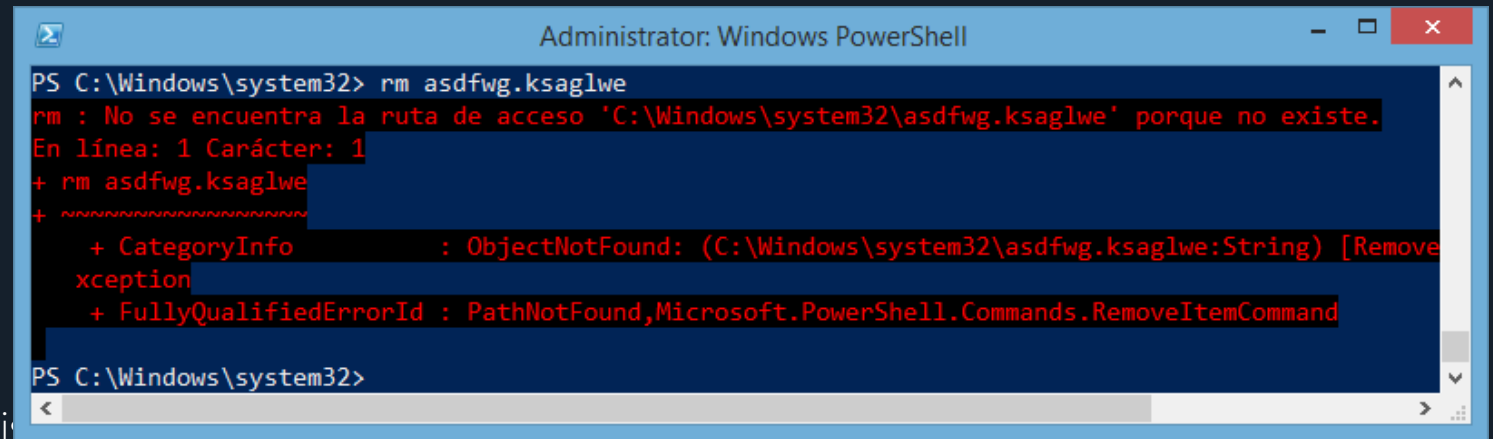
Control de errores

■ Dos tipos:

- Errores terminales: Finalizan la ejecución del script. Ej: Nulls
- Errores no-terminales: Dan un aviso de error, pero el script sigue funcionando. Ej: File-not-found

■ Variable \$error

- Almacena los errores encontrados.
- Es Global
- Tiene más de lo que se ve a simple vista (Get-Error, \$?)



```
Administrator: Windows PowerShell
PS C:\Windows\system32> rm asdfwg.ksaglwe
rm : No se encuentra la ruta de acceso 'C:\Windows\system32\asdfwg.ksaglwe' porque no existe.
En línea: 1 Carácter: 1
+ rm asdfwg.ksaglwe
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\Windows\system32\asdfwg.ksaglwe:String) [RemoveItemCommand]
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand
PS C:\Windows\system32>
```

GESTIÓN DE ERRORES

Bloque try..catch

- Misión: Capturar los errores terminales en el momento de producirse
- Añade robustez al código
- Permite recuperarse del error
- Finally es un bloque opcional que se ejecuta *siempre*, independientemente de si hubo error o no
- Para capturar errores no terminales, marcar \$erroractionpreference como **Stop**
- Se pueden capturar excepciones específicas

plain concepts

```
1 $Computer='notonline'
2 Try{
3     $os=Get-Wmiobject -ComputerName $Computer -Class Win32_OperatingSystem
4     -ErrorAction Stop -ErrorVariable CurrentError
5 }
6 }
7 Catch{
8     Write-Warning "You done made a boo-boo with computer $Computer"
9 }
```